Diploma Thesis

«A single threaded implementation of RT-SENMOS»

Savvaki Paraskevi

Supervisor: George Xylomenos

Athens, July 2015
Table of Contents

Protocol Description..................................................................................................................2
The need for single threaded programming.............................................................................6
Code Description.........................................................................................................................9
Performance evaluation.............................................................................................................13
References.................................................................................................................................18
Abstract

The goal of this project is to offer an implementation of RT-SENMOS protocol [1], using a single thread instead of multithreading. The implementation is a different version of the existing project, using channels instead of sockets. The main logic of the implementation will be analyzed in this report as well as a description of Java channels and their use. A performance comparison of the two implementations concludes the deliverable.
Protocol Description

In RT-SENMOS [1], communication between network elements is carried out in stages. In this section, we describe the five steps of communication between the sink and the sensors:

1) Connection establishment
2) Sensor information exchange
3) Data exchange
4) Idle period
5) Connection release

Connection establishment

The RT-SENMOS model defines two separate communication channels for each sensor, one for control data and one for user data. A separate control channel using a well-known UDP port is used for connection requests. When the sink begins to operate, it listens to a well-known UDP port for connection requests from the sensors. The sensors wait for the sink to become reachable before trying to connect to it. Each sensor periodically sends a probe message, MSG_HELLO, to the well-known IP address and UDP port of the sink until it receives a response. It is noted that the probe interval is configurable. Moreover, the routing protocol used may notify the sensor when the sink becomes reachable. The MSG_HELLO message includes the sensor identifier, which is unique to the sensor network in case the network nodes are pre-configured, and its type, i.e. event or continuous sensor. The sink responds to the MSG_HELLO message with a MSG_HELLO message. The MSG_HELLO message includes a dedicated UDP port, which will be used by the sensor for the control messages. The sink also measures the round-trip time (RTT) for that sensor.

Sensor information and idle

After receiving a response from the sink, the sensor prepares an MSG_INFO message which includes the delay between receiving the message from the sink and sending the response, the data rate requested by the sensor, whether the sensor is ready to send data at this point in time or not, the data packet size and the total size of the data to send. When the sink receives the MSG_INFO message it calculates the time elapsed since sending the MSG_HELLO, subtracting the delay in the message to get the RTT.
to the sensor. At this point, the connection has been established and the sink is aware of the sensor’s bandwidth requirements. If the sensor has indicated that it is not ready to send data at this point, the sink moves to an idle state, waiting until the sensor sends a MSG_INFO indicating it has available data to send. Then, the sink starts the data exchange by sending an MSG_CTRL_START message to the sensor, indicating the data rate to use. If the sensor indicated in the MSG_INFO message that it was ready to send data, the MSG_CTRL_START message is sent immediately as a response.

![Diagram of data exchange](image)

### Data exchange

After the sink sends the MSG_CTRL_START, the actual data transfer begins, using the UDP data port assigned for the transfer. The sensor breaks down its transmission into packets with the size indicated in the MSG_INFO message, until all the data indicated in the MSG_INFO message is exhausted. Data packets only have a single header field, a segment number used to sequentially number all data packets. When a missing packet is detected, the sink sends a MSG_NACK to the sensor. However, the sensor does not immediately retransmit lost messages. After the transmission is complete, all missing messages are retransmitted, generating further NACKs from the sink, if needed. This procedure is repeated in rounds, until all messages are received. Once recovery is complete, the sink sends a MSG_CTRL_DONE message to indicate a successfully completed transfer. If needed, the sink will send a MSG_CTRL_RATE message to the sensor indicating its new rate allocation. Note that round-based recovery allows the sink to use any packets received without waiting for retransmissions. The sink may even stop the recovery process by sending the MSG_CTRL_DONE message. For example, when an image is transmitted using redundancy coding, the sink may stop the recovery process when enough packets have been received to adequately reconstruct the image. This allows the application to fine tune the reliability of the protocol.
Connection control and release

Since the path between the sink and the sensor may become disconnected due to the sink’s mobility, the connection may fail between data transfers, without either side noticing. For this reason, the sink periodically sends an MSG_CTRL_ALIVE message to the sensor, which is acknowledged by an MSG_CTRL_ACK message from the sensor that includes the delay incurred between receiving the MSG_CTRL_ALIVE and responding with the MSG_CTRL_ACK. In addition to confirming that the connection is still alive, this procedure allows the sink to periodically measure the RTT of the connection. If the sensor or the sink wishes to complete the connection, they can send a MSG_CTRL_BYE message, which does not need to be acknowledged, as after either side drops the connection, the other one will eventually timeout: the sink times out if no responses are received to its MSG_CTRL_ALIVE messages, while the sensor times out if no MSG_CTRL_ALIVE messages arrive. The MSG_CTRL_BYE message simply allows the other end to release the resources dedicated to the connection without waiting for a timeout. The connection control messages are shown in Figure 4.
**Protocol Description**

**Congestion control**

One of the most important features of the protocol that was imported in this project is the congestion management implementation. The congestion management mechanism of the protocol is agile and purely sink-driven. The congestion is focused around the sink, and total available bandwidth is known as it depends on the technology used by the sink and sensors for data exchange. At first, a fixed part of this bandwidth is being reserved, e.g. 30%, for the control message exchanges which are not rate-controlled. Then, the sink splits the remainder to event and continuous sensors using a ratio determined by the application, e.g. 10%-90%, depending on the number and type of sensors present at the disaster site.

The congestion management algorithm periodically evaluates the state of individual connections and the system as a total. The sink monitors the RTT of each connection using the MSG_HELLO_ACK, MSG_INFO, MSG_CTRL_ALIVE and MSG_CTRL_ACK messages; the processing delay at the sensor is always subtracted to get an accurate RTT estimate. The congestion management algorithm maintains the last few RTT samples and their moving average.

Whenever the algorithm runs, it first checks whether the average for each sensor has increased compared to the previous value by more than a configurable threshold. If this occurs four times in a row, then the corresponding connection is congested, otherwise it is not. If the connection is congested, the sink instructs the sensor to reduce its rate by 20%, via a MSG_CTRL_RATE message. After each individual sensor is checked, if a new sensor has been connected or an existing one has been disconnected, the entire system is checked to see whether global adjustments need to be made. This takes places separately for each sensor class. First, the total rates requested (not assigned) by the sensors of the class are being calculated. If these are below the available bandwidth, they will all get what they asked for. New sensors will get their requested rate in the MSG_CTRL_START message which directs them to start sending data. Sensors that were previously rate limited, will increase their rate by 20%, while other sensors will get their requested rate; in both cases, the change is announced via a MSG_CTRL_RATE. If, on the other hand, the requested bandwidth is higher than the available one, the available rate is shared equally among all sensors of that class. The sensors are notified as above, i.e. either via a MSG_CTRL_START or via a MSG_CTRL_RATE message.

The congestion management algorithm is very simple, as congestion is expected to be concentrated around the sink. Since the sink is constantly on the move, it is very unlikely that a distributed congestion control management will have time to converge. While TCP uses an Additive Increase - Multiplicative Decrease (AIMD) algorithm, the scheme uses fixed and symmetric steps. This is because TCP sources constantly probe the network for capacity, hence entering deep into the congested region before having to abruptly backoff. In the scheme sensors are conservatively rate controlled, hence congestion is expected to appear slowly, therefore there is no need for dramatic rate reductions.
The need for single threaded programming

Our goal is to convert an existing project that uses socket connections from multithreaded to singlethreaded. The protocol described above presupposes sensors, but does not put restrictions on the number of sensors that may be present in the location that the sink enters. Therefore, there is a risk that the sink may not have the necessary memory to handle the load. Each thread add some overhead and the memory savings by removing threads can be valuable in these cases.

In general, the number of threads used is proportional to the memory used. Depending on the size and complexity of the program, this may cause an out of memory exception, especially if on a 32-bit operating system. In many cases the overhead of creating threads is not worth the useful work performed by them. Also, if we run easy work in threads, the program can be slower than with one thread. Trivial code is better ran in one thread.

In our project, we want to detect and study improvements on the runtime of the RT-SENMOS protocol. For this purpose we used the NIO library. Java NIO is an alternative IO API for Java. Unlike the common IO API which works with streams, the NIO uses channels and buffers. The data is read from a channel to a buffer or written from a buffer to a channel. It uses a thread which calls a Selector. The NIO Selector is a component which audits the channel to see whether there are data to be written or read.

Threads do not make the computer run faster by themselves. All they can do is increase the efficiency by splitting work among the cores, thus performing many instructions at the same time. But, if our system has a single core this is not useful to us. Generally, while NIO is not automatically always faster than plain IO, some operations are potentially faster using NIO especially when we have many threads that cost CPU time. NIO has better scalability than plain IO, especially when there are many clients. With NIO we can scale to many network connections much easier, because we do not need one thread per connection as with plain IO.

Channels and Buffers

Java NIO is Non-blocking, which means that the thread can do something else while the channel reads or writes to a buffer. With a single thread, the server can communicate with a number of open connections. To achieve this, DatagramChannels or SocketChannels are added to the Selector, and then with an iterative call of the select() method the server is informed about what connections are open, waiting or have sent data. This allows us to communicate with multiple clients within a single thread. The positive element of the singlethreaded programming is that we do not have overhead, which becomes greater as the number of threads is increasing, and we do not need to worry about synchronization.

In our project, a thread is created in the main method, which will be used for communicating with each new client. Then each client will communicate with a unique channel that will be created individually for this client and will be assigned a new port.

For this project we used DatagramChannels, so that the channel can send and receive
UDP packets. Their use is as follows:

To open a DatagramChannel and define that it can receive from a particular port:

```java
DatagramChannel channel = DatagramChannel.open();
channel.configureBlocking(false);
channel.socket().bind(new InetSocketAddress(port));
```

To access the data sent from the clients and in order to be able to send data, buffers are used to avoid the overhead created if one were to copy the data to a new array every time. To receive data we use the receive() method and we use ByteBuffers to store the data. Before calling receive(), we must call the clear() method, so that the buffer will not contain bytes from previous packets sent by the client. If a packet is larger than the size defined by allocate(), the remaining bytes will be discarded without interrupting or being noticed by the program.

```java
ByteBuffer buf = ByteBuffer.allocate(1024);
buf.clear();
channel.receive(buf);
```

In order to send data we call the send() method. We can write data to the buffer using the method put(). The method flip() changes the status of the buffer from reading mode to writing mode and vice versa.

```java
String msg = "data";
ByteBuffer buf = ByteBuffer.allocate(1024);
buf.clear();
buf.put(msg.getBytes());
buf.flip();
int bytesSent = channel.send(buf, new InetSocketAddress(ip, port));
```

Selector

We can create a Selector by calling the method open().

```java
Selector selector = Selector.open();
```

Then we need to connect the channel created above with the selector.

```java
SelectionKey key = channel.register(selector, SelectionKey.OP_READ, new ClientRecord());
```

To read from a channel, it must be in read mode – respectively, it must be in write mode in order to write to it. This is achieved by the second parameter, where
SelectionKey.OP_READ is used for reading and SelectionKey.OP_WRITE for writing. The above line of code means that with the creation of the selector we will first wait to receive data from a client and then we can continue with the data processing.

The register() method returns a SelectionKey. Each key in the SelectionKey is linked to an incoming message from the client. All channels must be registered to the same selector in order to be monitored properly.

The last argument of register() is used so as to be easier to edit the key. The key is connected to an object of the ClientRecord class. This class contains an integer field called ECHOMAX where we store the maximum size that a packet from the client can be. This is useful, because we don't want byte losses, as it would happen if we allocated less bytes to the ByteBuffer. Furthermore, the class contains a SocketAddress object which will be used to store the IP and port of the client who sent this particular message and a ByteBuffer for the data.

The keys are the objects used by the selector to sort the client requests. Each key represents a single request from a client so the next request from the same client will not have the same key with any that has come before from this client or another. The keys also contain information to distinguish this client from the rest.

Afterward, using a while loop in main(), the selector will wait and manage the data. The method selector.select() returns the number of channels that have received data from a client. If the number is not equal to zero, then the keys that exist in the selector are stored in an iterator, so that for each key we will do a read and/or a write. Each time a client wants to start a communication with the server, the client must send a message, the server must read it, send the appropriate answer then the client read it and start a new round accordingly. Before we can take the next key from the iterator we must delete the one we already processed because it has no further use.

If the key is readable the handleRead() method is called, otherwise if it is writable we call the handleWrite() method.
Code Description

Server

From the existing codebase, classes Log, TxtLogger, Configuration, DataPacket, RTTMeasurement, PreventionCongestion and SensorPool were used without any changes. We added class Sensor to keep the characteristics of each client, such as the id, type, IP address and also the state of the client so as to know what to expect in the contents of the message. Furthermore, the changes made to SinkMain are described below.

SinkMain

In order to keep track of all clients with whom there is communication, there is an ArrayList called sensorArray, which stores Sensor type objects. There is also the variable arrayLength where the user can define the number of sensors the server can handle in parallel. For example, if arrayLength has the value 1, the server must first complete the communication with the first client that came to the server, before it can receive messages from any other client.

In the main method, the procedure to create the channel and selector described above takes place. There is a possibility, however, that some client sent messages but for some reason disappeared from the scope of the server. In order to avoid the risk of such clients remaining in the sensorArray, before calling the selector, we check the time we last saw a message coming from this client against the current time and if the difference is greater than 2 minutes, the client is removed from the ArrayList. We also call the recomposeData() method in order to reconstruct whatever messages have been received from this client. For each client we call the handleRead() method and then the handleWrite() method if needed.

HandleRead()

The method handleRead() takes as argument a key. We take the channel connected to this key using the method key.channel(), while the command key.attachment() takes the ClientRecord object associated with the key. The receive() method returns the SocketAddress object linked with this particular message, and the contents of the buffer are stored in a String called msg. Firstly, we check if the sensorArray already has this client comparing the SocketAddress. Then we check the contents of the message msg.

If the message contains MSG_HELLO, we extract the id and type and store them in a new Sensor object with state NEGOTIATING and the SocketAddress extracted before. If the sensorArray is not full and this Sensor is not already in it, we add it. Otherwise, an appropriate message appears. Depending on the type of the client, it is stored as an event or a continuous sensor in SensorPool. Moreover, for each new client we construct a new channel which is registered to the selector and is bind to a port to be used exclusively by that client.

For the rest of the messages of the client there must be a Sensor object with its
SocketAddress in the sensorArray, which means it must have sent at least once (MSG_HELLO). If we are in this case, and we receive MSG_CTRL_ACK then we measure the RTT for this client. If we receive MSG_BYE the state becomes EXITING. In both cases we do not need to send anything in response to the client so the key remains in read mode.

If we receive MSG_INFO and the sensor expects RECEIVE_SENSOR_INFO for this client, then it measures the RTT again and saves the information we got from the message to the Sensor object. This information also tells us if there are available data to be sent from the client. If there are, the congestion algorithm is called. As we have received information about the size of the packets that will be sent, we update the ECHOMAX field. Finally, the sensor is put in GETDATA state. Otherwise, if no data are available, the sensor enters the IDLE mode.

If the message we received did not fall in any of the previous cases, and the sensor’s state is GETREST, then we know we must expect to receive the file. First, the method deserialize() is called, which returns an object of type DataPacket. The DataPacket will contain the segmentID and the file to be written to the server. If the bytes received by the server so far are less than the file size, which means we have not finished receiving packets, we check the segmentID to see which packet we received this time. The Sensor object contains a HashMap with key the segmentID and value the bytes. If the segmentID exists in the HashMap it means the server has already received and saved this packet.

If the segmentID is greater than the one expected, we detect packet loss. To find all the packets that have been lost since the last one received, we use a while loop. In order to store the NACKs that must be sent to the client we create a List which will contain the control messages. Also, each sensor has a HashMap storing the segmentIDs of the packets that got lost along with the message that must be sent to the client in order to inform him of the loss. Another HashMap stores the segmentIDs with value false if the NACK was transmitted in this cycle of retransmissions. It is possible to have many cycles of retransmissions because for big packet loss the packets that get lost are so many that can be lost again when retransmitted. Using the above two HashMaps we can see which NACK messages must be sent immediately.

The reason why we use two data structures in order to store the missing packets is that we want to separate the case where the NACKs are sent for the first time and we have not received any retransmitted packets, from the case where we have received retransmitted packets and we must do further controls in order to extract the NACKs that should be sent again.

If the packet has segmentID smaller than what we expected and the segmentID does not exist in the HashMap with all the received packets, it means we received a retransmission of a previously lost packet. If the segmentID of the packet is in the HashMap that contains the packets for which a NACK message has been sent, then for all the packets with smaller segmentID in this HashMap, we set the value to false since we are sure the NACK messages associated with them have not been received by the client and so we must send them again. The packet we received is stored as removedID and is removed from the HashMap that contains the NACKs.

When the bytes read by the server for this sensor are equal to the file size, the method recomposeData() is called.
If in any of the above cases it is required to send data to the client, before handleRead() finishes, the key changes to writing mode using the command key.interestOps(Selection.OP_WRITE).

After the handleRead() ends, the thread returns to the main() where if the key is set to write mode, the handleWrite() method is called. Otherwise, we delete the key and the next one is called.

HandleWrite()

It takes a key as an argument in order to extract the recipient of the data to be sent. In case the server must sent more than one messages in a row to the client before the client can send again, at the end of the message the server adds 1 if more messages follow or 0 if this message is the last. If this is not a case of multiple transmission, nothing is added to the message.

If the state of the sensor is NEGOTIATING, the server sends an MSG_HELLO message and the port which the client must use to send any further messages to the server. Also, we store the time when the transmission took place as delay in the currentSensor object so as to use it afterwards to calculate the time passed. We change the value of the state field into RECEIVE_SENSOR_INFO.

If the state is IDLE, which means that the client did not have data available to send, the server sends MSG_HELLO to the client to let him know that that connection with the server is not lost. The state's value is updated to RECEIVE_SENSOR_INFO as the server will now wait data from the client when they are available.

If the state is GETDATA a MSG_CTRL_START message is sent to the client along with the rate to be used by the client. This message means that the server will now expect to receive the file. The state is changed to GETREST.

If the state is GETREST it means that the transmission of the file has started.

If there are control messages, they are sent using a while loop.

If there are no control messages there are two possible cases. In the first one, we need more than one cycle of retransmissions so if there are segmentIDs for which, for this sensor, we should have received retransmissions before the last packet we received, and for this cycle of retransmissions we have not send NACKs yet, we save the NACKs into an ArrayList and we update the HashMap defining that we sent NACKs for these packets. Afterwards, we send the contents of the ArrayList. In the second case, we do not need to send a message or a NACK to update the rate so we just send MSG_CTRL_ALIVE.

If the state is equal to EXITING, a MSG_CTRL_DONE is sent to the client and the sensor is removed from the SensorPool and the sensorArray.

Client

The client uses sockets rather than channels, as it was not necessary to change its structure in order to communicate with the server. Classes Logger, TxtLogger, Configuration and DataPacket remained the same. The changes is SensorMain are explained below.
SensorMain

In order to know when the sink is reachable, a MSG_HELLO message is sent repeatedly for as long as we do not receive an answer. When a MSG_HELLO message is received from the server we extract the port which will be used for any further transmissions. The port is saved in the dstCtrlPort field of the Configuration file. The client then sends a MSG_INFO packet and if it receives MSG_CTRL_START, the client starts sending the file into packets. After each transmission the client does a receive because the server will either send control messages so MSG_CTRL or/and MSG_CTRL_RATE messages will be received or it will not have any control messages and it will send MSG_CTRL_ALIVE. If a control message ends in 1, it means that more follow, so the receive method is called again. As in the project which uses sockets instead of channels, the packets for which the client received NACK are stored in order to be retransmitted.
Performance evaluation

In order to check the code with respect to the time needed to complete and its memory consumption, we had to simulate a real network with packet loss. For the simulations we used the netem linux tool. With the command

```
sudo tc qdisc add dev eth0 root netem loss 2%
```

we obtained 2% of packet loss. The packet loss was applied to the virtual machine where the client was located.

Example 1

For the simulation in this example we use the sink and one sensor. In both the socket project and the channel project, the client send an 8MB text file to the server.

**Time**

<table>
<thead>
<tr>
<th>Packet Loss</th>
<th>Channels</th>
<th>Sockets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>11 sec</td>
<td>118 sec</td>
</tr>
<tr>
<td>0.5%</td>
<td>14 sec</td>
<td>120 sec</td>
</tr>
<tr>
<td>2%</td>
<td>22 sec</td>
<td>121 sec</td>
</tr>
<tr>
<td>5%</td>
<td>27 sec</td>
<td>127 sec</td>
</tr>
</tbody>
</table>

To measure the time taken, we subtracted the time the client first send to the sensor from the time the transmission was completed. This can also be observed from the log files of either the server or the client.

The above chart shows the time it took both the projects to complete. The channel
Performance evaluation

The NIO contributes in making the I/O faster. The NIO moves the most time-consuming input and output activities back to the operating system. Thus, writing and reading are faster than the traditional I/O used in sockets.

The reason why we use multithreading is to make the program run faster by executing many tasks at the same time. In this respect, channels have a disadvantage as they use only one thread, so only one client can be served at a time. But channels can end up faster by completing the I/O operations in less time than the traditional IO. With big files as the one used in the experiment, the improvement is obvious.

We observe that even for zero packet loss, channels are much faster. As stated before there can be cases where sockets have better results than the channels. But in our experiments with the specific files that we used we have significantly better results when we use channels.

Memory

<table>
<thead>
<tr>
<th>Packet Loss</th>
<th>Channels</th>
<th>Sockets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>28 MB</td>
<td>50 MB</td>
</tr>
<tr>
<td>0.5%</td>
<td>30 MB</td>
<td>50 MB</td>
</tr>
<tr>
<td>2%</td>
<td>34 MB</td>
<td>60 MB</td>
</tr>
<tr>
<td>5%</td>
<td>35 MB</td>
<td>66 MB</td>
</tr>
</tbody>
</table>

For the measurement of the memory used by the programs, the following lines of code we used in the server:

```java
Runtime runtime = Runtime.getRuntime();
runtime.gc();
```

The first two lines of code are used at the beginning of the program whereas the next two as executed after the file has finished been written locally.

```java
long memory = runtime.totalMemory() - 
runtime.freeMemory();
System.out.println("Used memory is bytes: " + memory);
```
In general, we observe that the memory requirements are up to several MB, which makes sense as the program stores the packets it receives, so it will be at least 8MB, but it also stores control messages and messages in order to manage packet loss.

We observe that sockets are using a little less memory than the channels when we have little packet loss. But while the channels remain relatively stable in memory consumption, the sockets’ memory consumption greatly increases for larger packet loss.

With large packet loss, more weight is placed on the threads, as we have more NACKs. The messages sent to the client are more expensive in memory as they contain more information to identify the lost packets. Also, for every operation a new thread is created, adding additional overhead. In contrast, using channels where there is a single thread, no extra overhead is added.

For small packet loss the channels use a little more memory than the sockets. With channels we use more data structures in order to store the clients and their states and characteristics since there is no thread exclusively devoted to each client. Using only one client the sockets do not have much memory requirements, in order to save the thread overhead.
Example 2

For this example we used the sink and two sensors. One sensor send an 8MB text file as before and the second sensor send a 2MB zip file.

<table>
<thead>
<tr>
<th>Packet Loss</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>31 MB</td>
<td>14 sec</td>
</tr>
<tr>
<td>0.5%</td>
<td>34 MB</td>
<td>16 sec</td>
</tr>
<tr>
<td>2%</td>
<td>37 MB</td>
<td>24 sec</td>
</tr>
<tr>
<td>5%</td>
<td>39 MB</td>
<td>30 sec</td>
</tr>
</tbody>
</table>

As expected with two clients we need more time and memory. Both programs started at the same time.
Performance evaluation
References